Self-organizing Tuple Reconstruction in Column-stores

E0261

Jayant Haritsa

Computer Science and Automation

Indian Institute of Science

Motivation

- How can we make Tuple Reconstruction easier?
  - if columns are sorted in the same order
  - Eg: Discount BAT is sorted on Discount, and Day BAT is reorganized to have matching tuple-ID sequence

```
Select Day, Discount
From   Sales
Where Discount<=0.15
```

- However, the sorted tuple-ordering cannot always be preserved
  - During query processing, many operators (joins, group by, order by, etc.) are not tuple order-preserving

Database Tuning: A New Approach

Not enough space to index all the data

Not enough idle time to finish proper tuning

By the time we finish tuning, the workload changes

Contd …

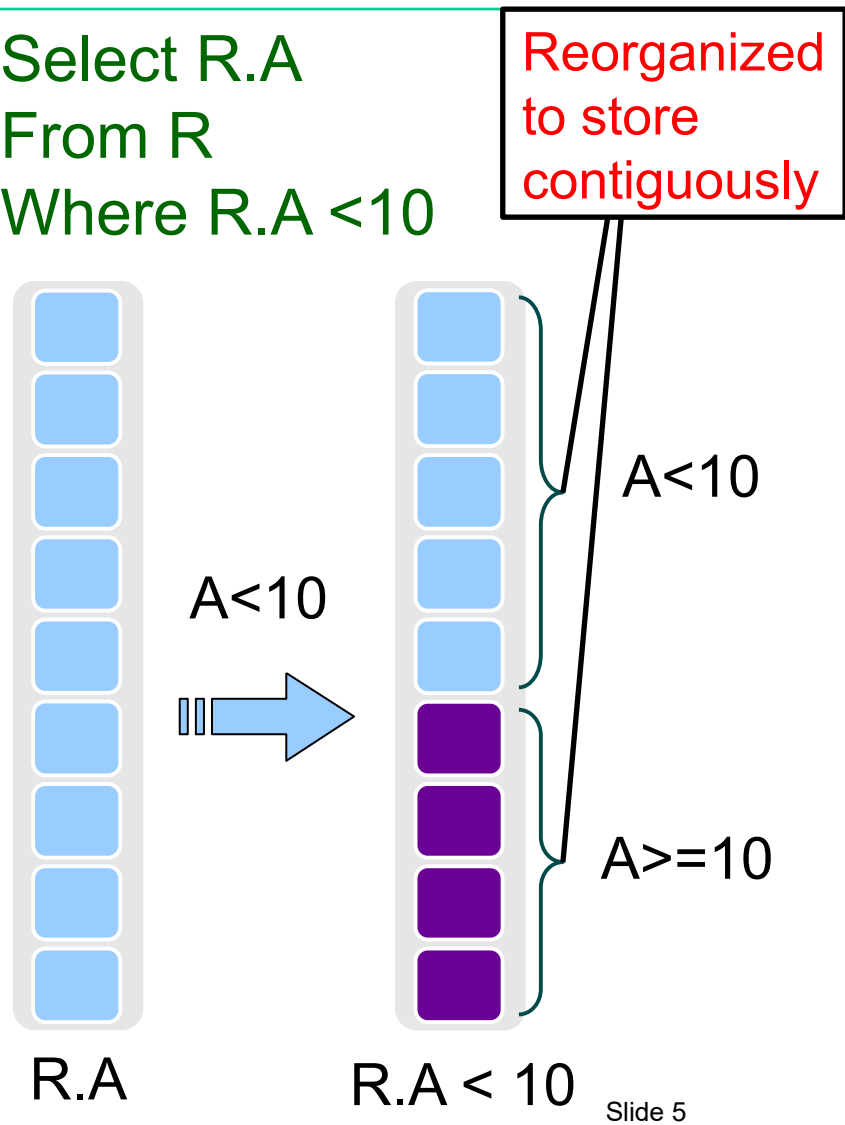
- What kind of indices will improve the data access?
- When should we create an index or drop an index?
- In which part of data should we create an index?

Database Cracking

Stratos Idreos won the 2011 ACM SIGMOD Jim Gray Doctoral Dissertation Award for his thesis 'Database Cracking: Towards Auto-tuning Database Kernels'.

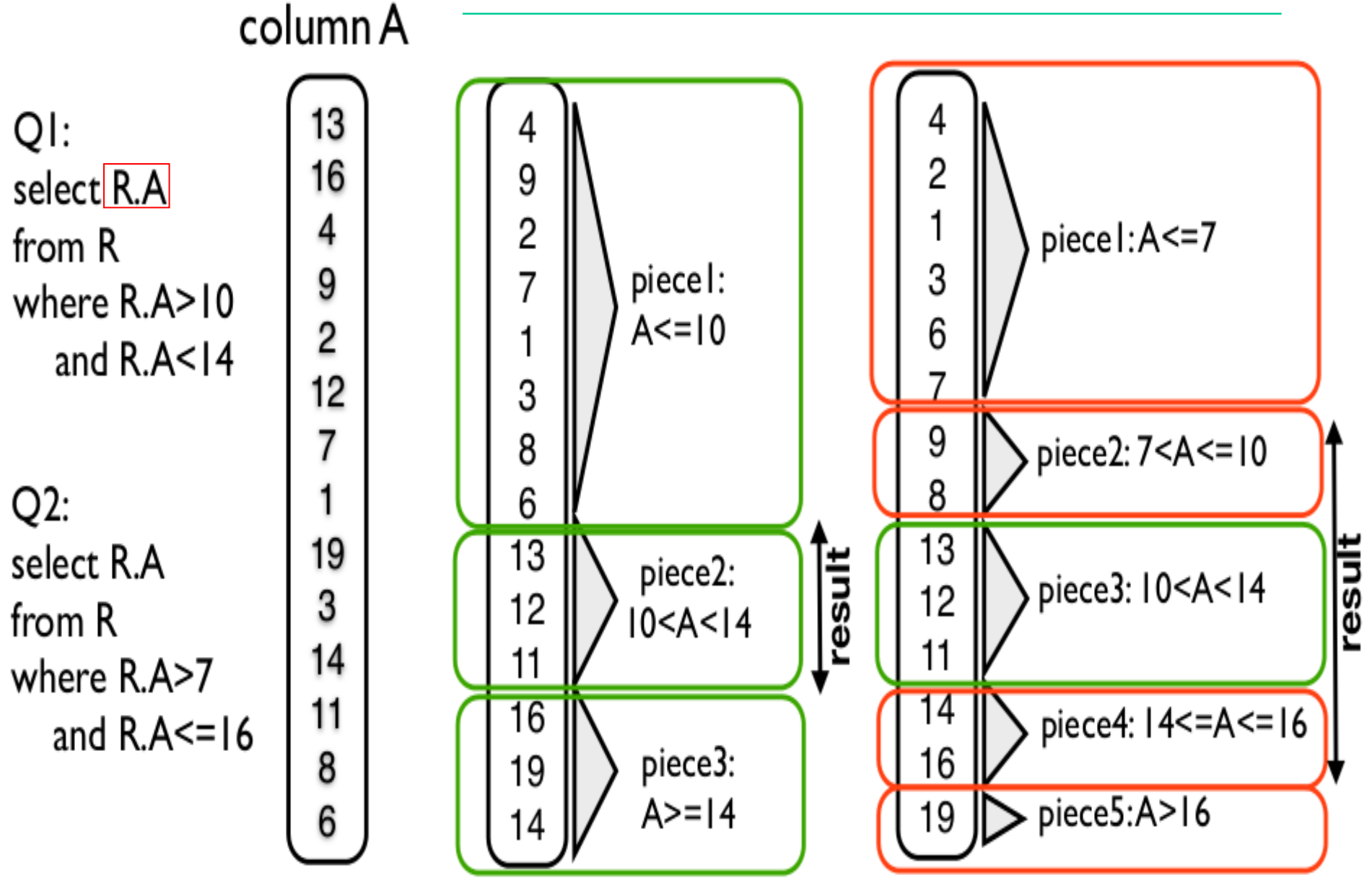
Database Cracking

- Self-tuning database kernel
- Every query is treated as an advice on how data should be stored
  - Physical reorganization happens per column based on selection predicates



Cracking: Example

Note: Data portion not used is not tuned



Selection-Based Cracking

- The first time an attribute *A* is required by a query, a copy of column *A* is created, called the cracker column C_A of *A*
- Each selection operator on *A* triggers a range-based physical reorganization of C_A
  - *Implemented similar to bubble-sort across the beginning and end segments of the cracked portion* [Algo 2 of CIDR07 paper]
- Each cracker column, has a cracker index (AVL-tree) to maintain partitioning (piece) information
- Future queries benefit from the physically clustered data and do not need to access the whole column

The crackers.select Operator

crackers.select(A, v1, v2)

- First, it creates C_A if it does not exist
- It searches the index of C_A for the area where v1 and v2 fall
- If the bounds do not exist, i.e., no query used them in the past, then C_A is physically reorganized to cluster all qualifying tuples into a contiguous area
- Output:
  - A list of keys/positions

But, how to handle queries such as
Select R.B From R where R.A <10

Cracker Map

- For Query Type
  - Access attribute *B* based on attribute *A* of same relation R
- A cracker map M_{AB} is defined as a two-column table over two attributes *A* and *B* of relation *R*
  - Values of *A* are stored in the left column, called head
  - Values of *B* are stored in the right column, called tail
- Values of *A* and *B* in the *same* position of M_{AB} belong to the *same* tuple of R

Maps are Created on Demand Only

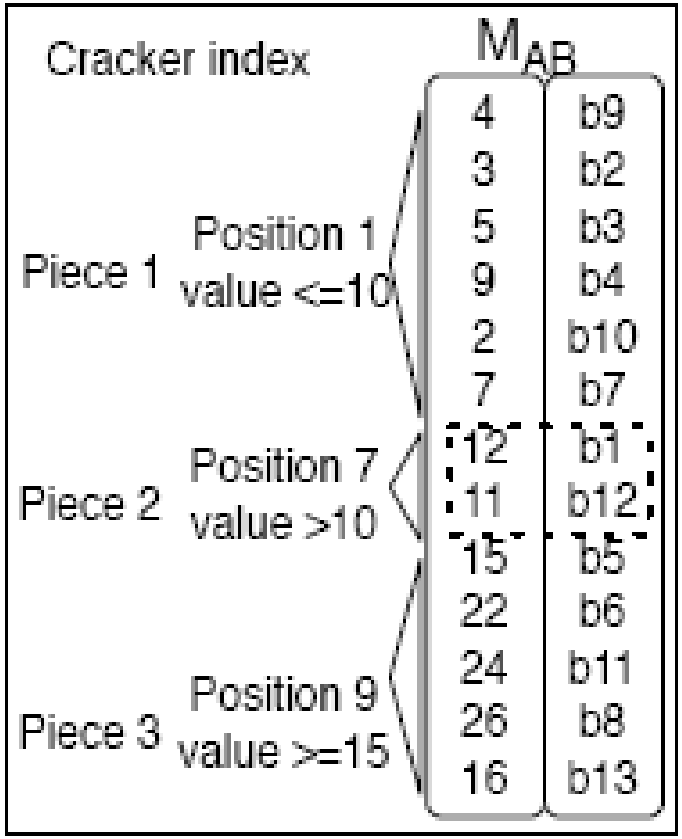
- When a query *q* needs access to attribute *B* based on a restriction on attribute *A* and M_{AB} does not exist,
  - then *q* will create M_{AB} by performing a scan over base columns *A* and *B*
- Else, if M_{AB} already exists,
  - All tuples with values of *A* that qualify the restriction are in a contiguous area in M_{AB}
  - Realized by splitting a piece of M_{AB} into two or three new pieces
- For each cracker map M_{AB}, there is a cracker index (AVL-tree) that maintains information about how *A* values are distributed over M_{AB}

Example

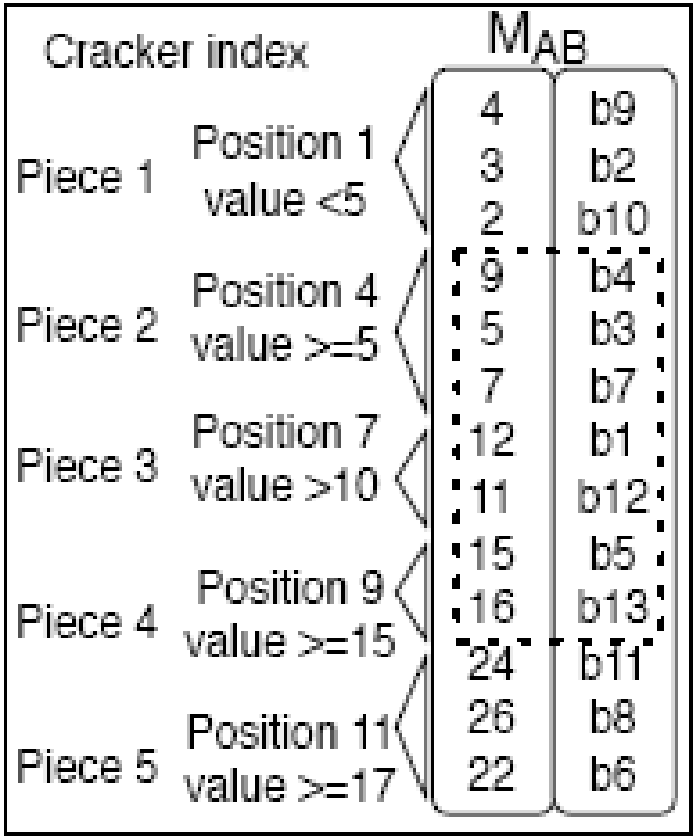
Initial state

A	B
12	b1
3	b2
5	b3
9	b4
15	b5
22	b6
7	b7
26	b8
4	b9
2	b10
24	b11
11	b12
16	b13

select B from R where 10<A<15



select B from R where 5=<A<17



Sideways.select(A, v1, v2, B) Operator

Returns tuples of attribute *B* of relation *R* based on a predicate on attribute *A* of *R* as follows:

(1) If there is no cracker map M_{AB} , then create one

(2) Search the index of M_{AB} to find the contiguous area *w* of the pieces related to the restriction σ on *A*

If σ does not match existing piece boundaries,

(3) Physically reorganize *w* to move false hits out of the contiguous area of qualifying tuples

(4) Update the cracker index of M_{AB} accordingly

(5) Return a non-materialized view of the tail of *w*

Multiple Projection Queries

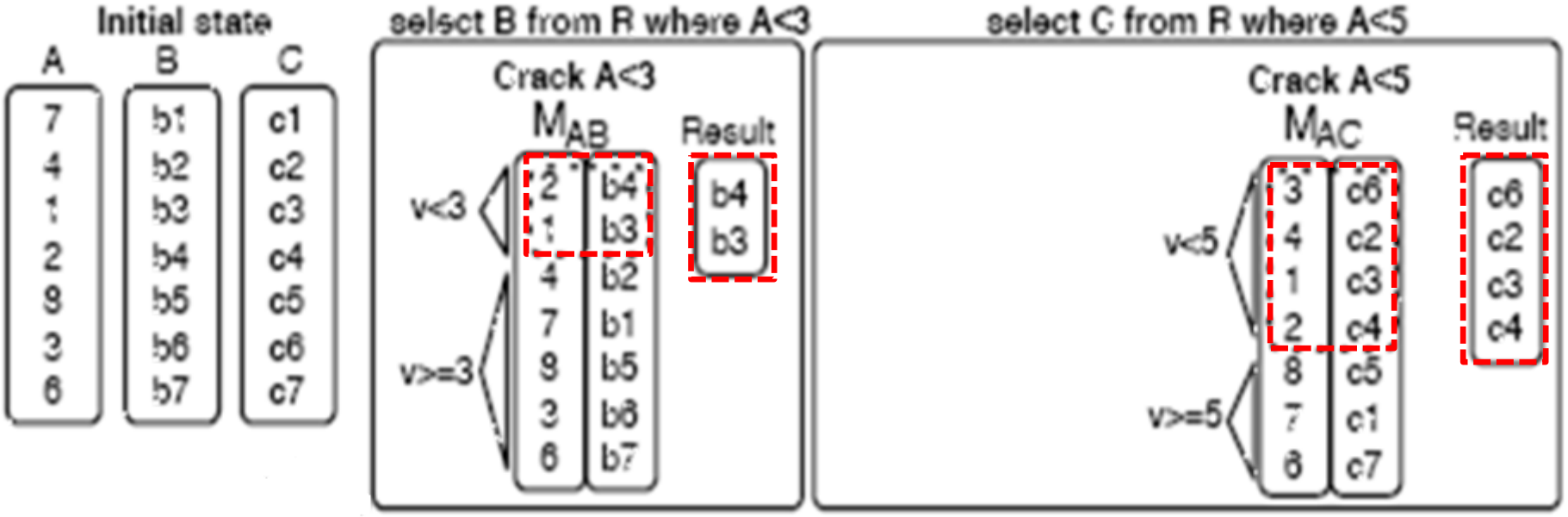
- How to handle the following query?

  Select B, C

  From R

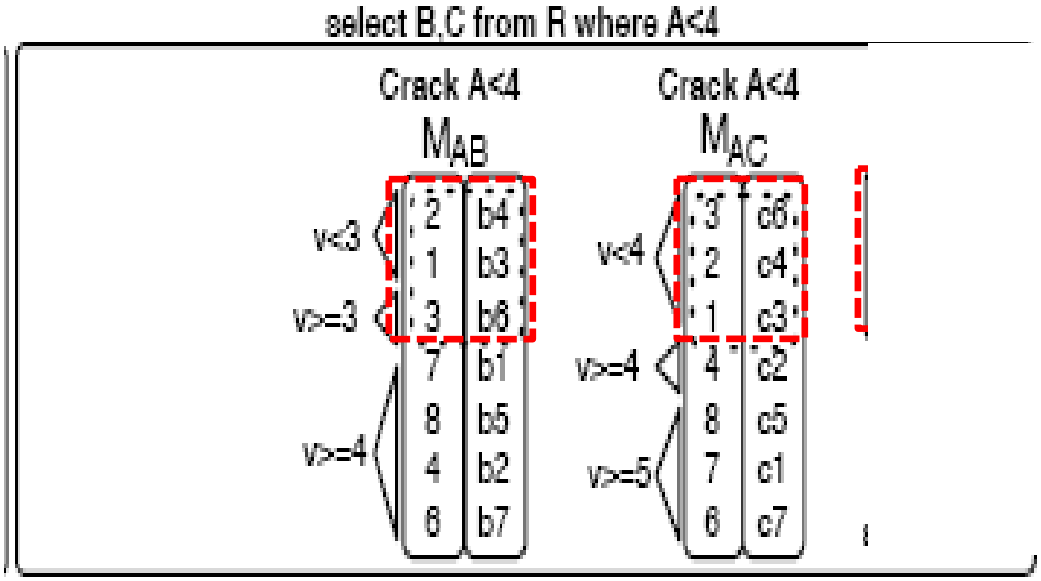
  Where A < 10

- For this query, we need 2 maps M_{AB} and M_{AC}
- In general, a single-selection query *q* that projects *n* attributes requires *n* maps, one for each attribute to be projected
- All maps that have been created using *A* as head are collected in the ***map set*** S_A



The Problem

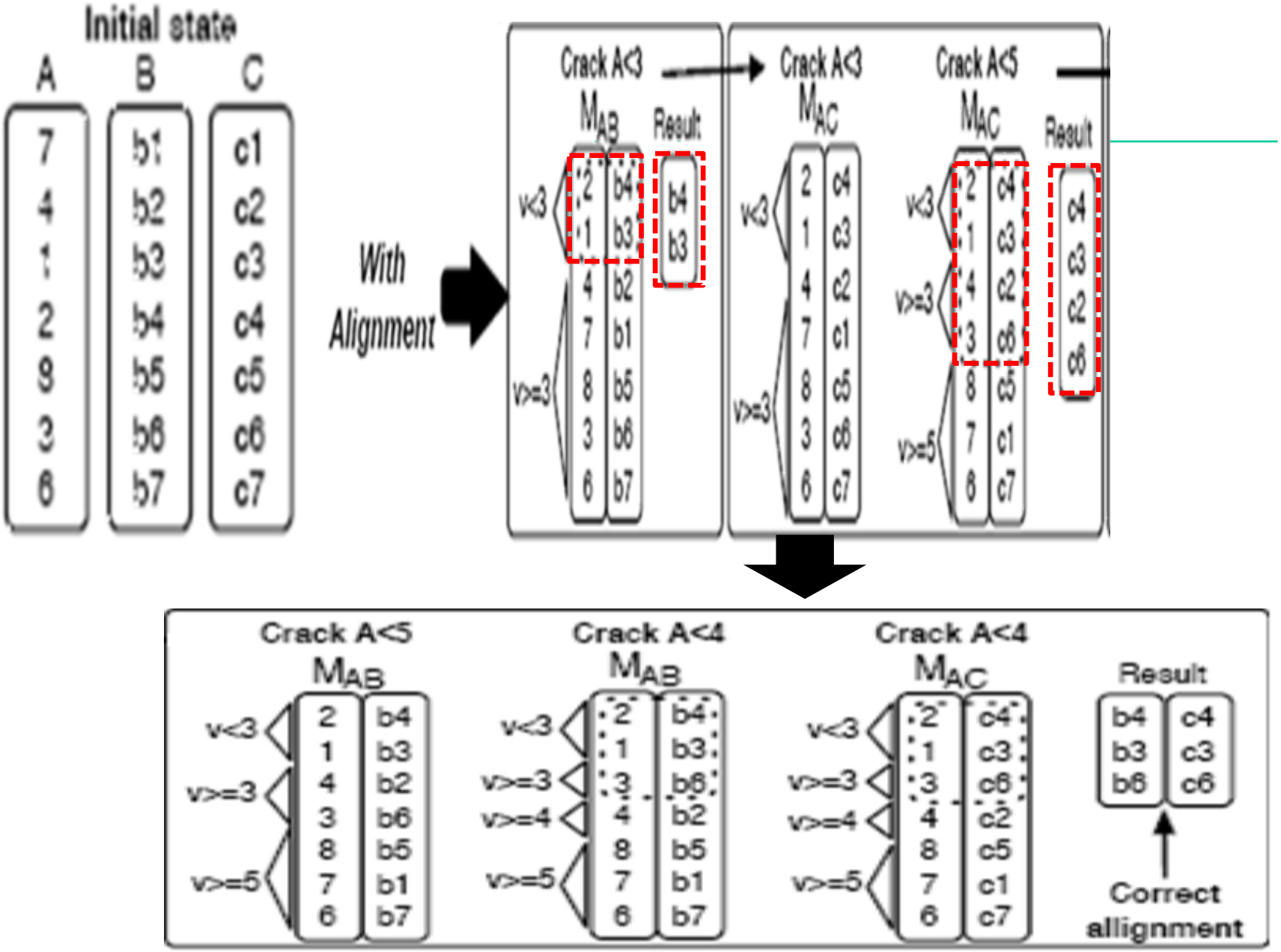
Naïve use of the sideways.select operator may lead to non-aligned cracker maps



Solution: Adaptive Alignment

- Extend the sideways.select operator with an alignment step to keep the maps aligned

- The Basic Idea
  - is to apply all physical reorganizations, due to selections on an attribute *A*, in the *same order* to all maps in the ***map set*** S_A


Initial state
A
B
C
With Alignment
Crack A<3
Crack A<3
Crack A<5
Result
Crack A<5
Crack A<4
Crack A<4
Result
Correct allignment

Cracker Tape

- For each map set S_A, introduce a cracker tape T_A
  - T_A logs (in order of their occurrence) all selections on attribute *A* that trigger cracking of any map in S_A
  - Each map M_{Ax} is equipped with a *cursor* pointing to the entry in T_A that represents the last crack on M_{Ax}
- Given a tape T_A , a map M_{Ax} is aligned (synchronized) by successively forwarding its cursor towards the end of M_{Ax}
- And incrementally cracking M_{Ax} according to all selections it passes on its way
- All maps whose cursors point to the same position in T_A , are physically aligned

The Extended sideways.select Operator: Summary

(1) If there is no T_A, then create an empty one.
(2) If there is no cracker map M_{AB}, then create one.
(3) Align M_{AB} using T_A.
(4) Search the index of M_{AB} to find the contiguous area w of the pieces related to the restriction σ on A.
If σ does not match existing piece boundaries,
  (5) Physically reorganize w to move false hits out of the contiguous area of qualifying tuples.
  (6) Update the cracker index of M_{AB} accordingly.
  (7) Append predicate $v_1 < A < v_2$ to T_A.
(8) Return a non-materialized view of the tail of w.

Multiple-Select Queries

Query: Select D From R
Where 3<A<10 and
4<B<8 and 1<C<7

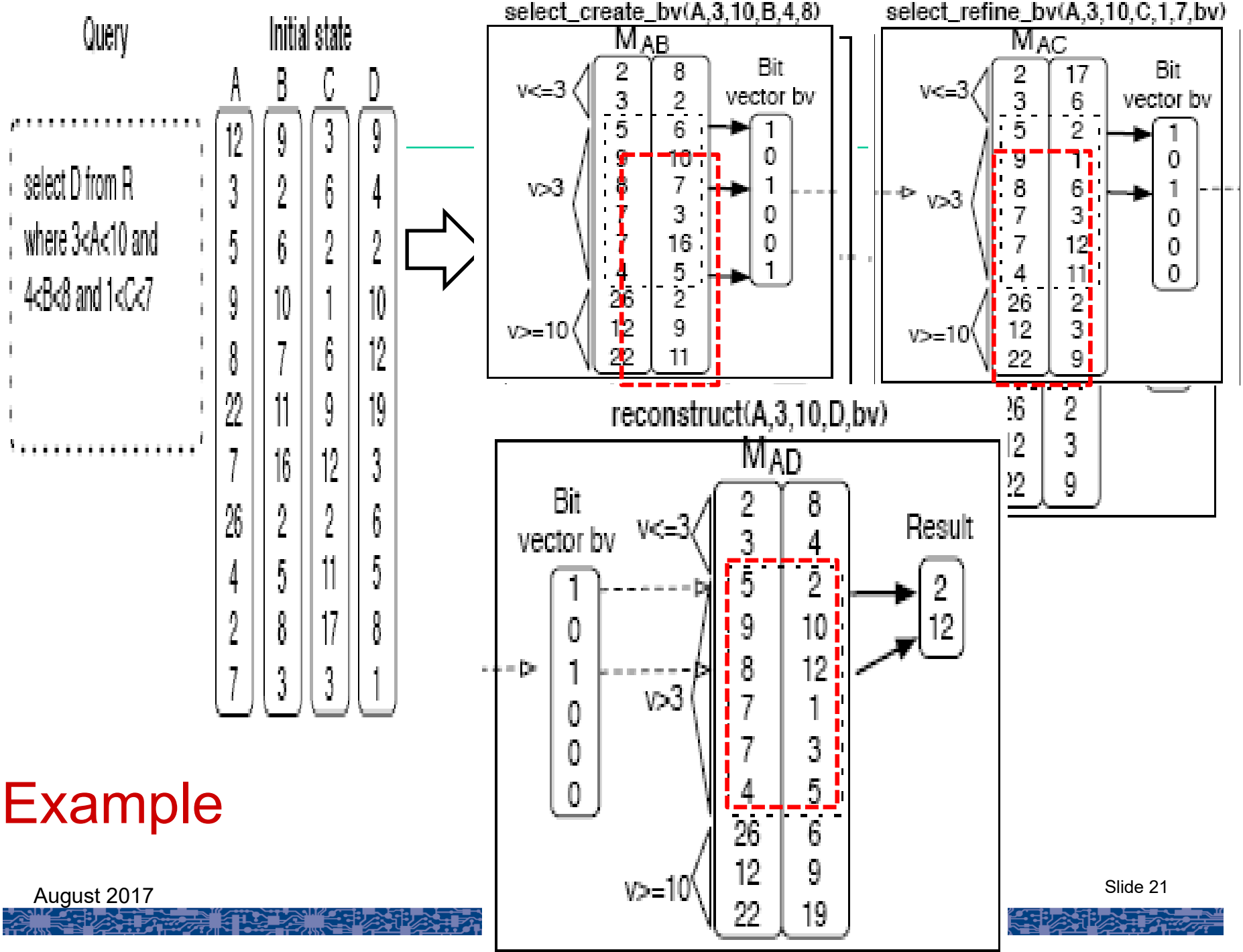
First-cut solution

- Create different maps $M_{AD,}$ $M_{BD,}$ M_{CD}
- Does this work?
- No, because of distinct sorting orders!

Nice Trick!

Query: Select D From R
Where 3<A<10 and
4<B<8 and 1<C<7

- Choose a selection predicate, say *A* (tuple ordering, which is essential, is possible only on one predicate)
- Create Maps $M_{AB,}$ M_{AC} *and* M_{AD}
- Since B is also a selection predicate in the Map $M_{AB,}$ create a bit vector for B which identifies the tuples satisfying predicate B
- Same is repeated for M_{AC}
- Use ANDing of the two bit vectors with Map M_{AD} to get the desired result


Query
Initial state
select D from R
where 3<A<10 and
4<B<8 and 1<C<7
A B C D
12 9 3 9
3 2 6 4
5 6 2 2
9 10 1 10
8 7 6 12
22 11 9 19
7 16 12 3
26 2 2 6
4 5 11 5
2 8 17 8
7 3 3 1
select_create_bv(A,3,10,B,4,8)
M AB
v<=3
v>3
v>=10
Bit vector bv
select_refine_bv(A,3,10,C,1,7,bv)
M AC
reconstruct(A,3,10,D,bv)
M AD
Result
2
12

Example

Map Set Choice: Self-organizing Histograms

- We had chosen predicate A randomly to be head for all the Maps. Is there a better choice of the predicate?
- Yes! for a query *q*, a set S_A is chosen such that the restriction on *A* is the most selective in *q*
  - Yielding a minimal-size bit vector
- The most selective restriction can be found using the cracker indices
  - choose the most aligned Map M_{AX} (for accurate statistics)
  - Estimate cardinality interval of the predicate using index boundaries

Complex Queries

- Apart from tuple reconstruction, other operators do not depend on tuple insertion order
  - Not affected by cracking physical reorganization
  - Joins, aggregations, groupings, etc.
  - Therefore, use standard column store operators

- Potentially many operators can exploit the clustering information in the maps
  - MAX operator can consider only the last piece of a map

Experimental Analysis

- Compare the implementation of selection and sideways cracking on top of MonetDB
- Against the non-cracking version of MonetDB
- And against MonetDB on presorted data
- Results
  - Sideways cracking achieves similar performance to presorted data
  - But does not have the heavy initial cost and the restrictions on updates and workload prediction

Partial Sideways Cracking

- Consider storage restriction
- Partial Maps
  - Maps are only partially materialized driven by the workload
  - A map consists of several chunks
  - Each chunk is a separate two-column table
  - Each chunk contains a given value range of the head attribute of this map
  - Each chunk is cracked separately

Partial Map Example

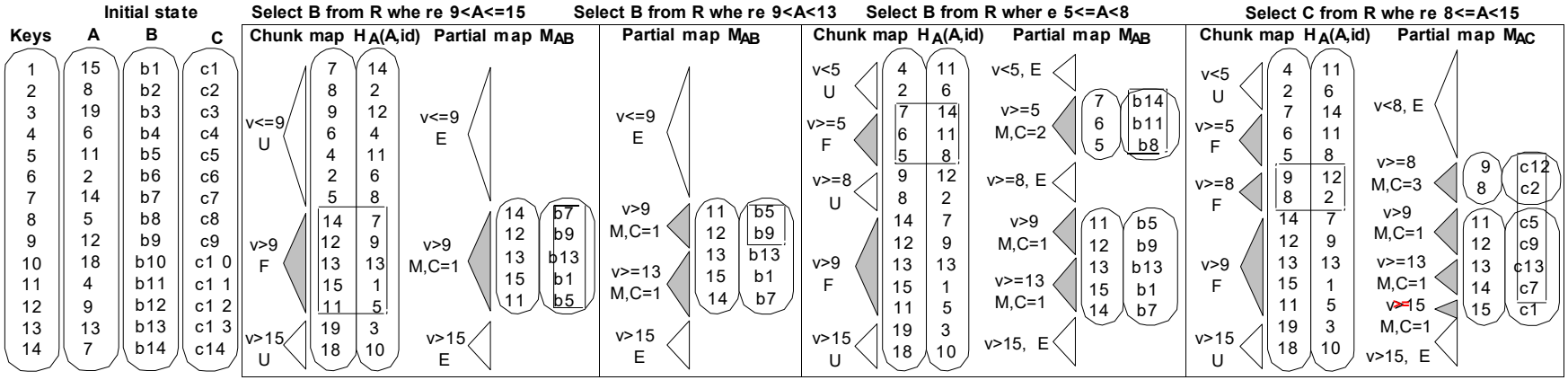


Figure 8: Using partial maps (U=Unfetched, F=Fetched, E=Empty, M=Materialized, C=ChunkID)

END